escuelaingenieríaindustrial
PONTIFICIA UNIVERSIDAD CATOLICA DE VALPARAISO

Métodos de ordenamiento y búsqueda para datos en memoria principal

Franco Guidi Polanco
Escuela de Ingeniería Industrial
Pontificia Universidad Católica de Valparaíso, Chile
fguidi@ucv.cl

Actualización: 12 de mayo de 2005

Ordenamiento

- Dada una colección que contiene un número elementos, el ordenamiento es el proceso de reacomodar un conjunto de elementos dados en un orden específico.
- Propósito: facilitar la posterior búsqueda de elementos dentro del conjunto ordenado.
- Tomaremos como estructura de referencia el arreglo.

Nota: Se agradece a Marcelo Silva F. el material presentado en esta sección

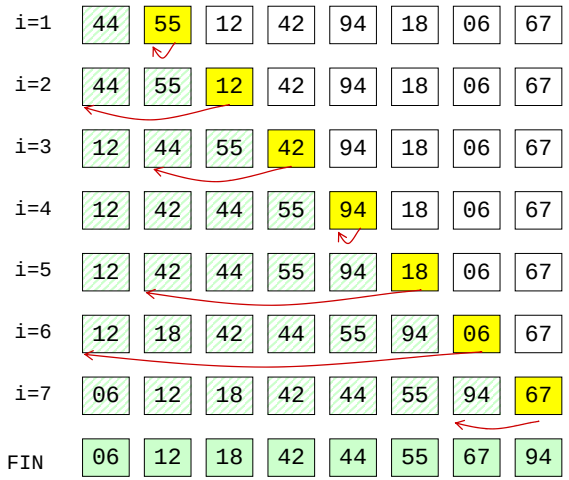
Ordenamiento en memoria principal (arreglos)

- Los elementos deben ser ordenados “in-situ”, es decir, el ordenamiento debe basarse en un intercambio de elementos dentro del arreglo, sin utilizar un segundo arreglo o estructura de datos.
- Métodos a analizar:
  - Inserción
  - Selección
  - Burbuja
  - QuickSort

Ordenamiento por inserción

- Similar a método usado por un jugador para ordenar una mano de cartas.
- Los elementos (cartas) son divididos conceptualmente en un conjunto de destino ($a_0..a_{i-1}$) y uno de origen ($a_i..a_n$).
- En cada paso, comenzando con i =1, e incrementando i en una unidad, el a_i es elegido y transferido al conjunto de destino, insertándolo en la posición adecuada.

Ejemplo de ordenamiento por inserción



Código para ordenamiento por inserción

```
static void inssort(Elem[] array) {
 for (int i=1; i<array.length; i++)
  for (int j=i; (j>0) && (array[j].key()<array[j-1].key());j--)
   DSutil.swap(array, j, j-1);
}
```

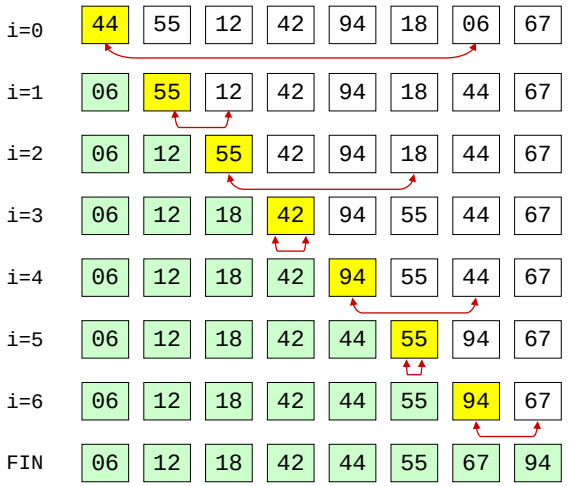
Notas:

1. Suponga que el método `DSutil.swap( Elem[] arr, int k, int l )` intercambia los elementos contenidos en las posiciones `k` y `l` del arreglo `arr`.
2. El método `Elem.key()` retorna un valor que será utilizado como clave de ordenamiento

Ordenamiento por selección

- En cada paso, comenzando con $i=0$, escoger el elemento con la menor clave en el conjunto $a_i..a_n$.
- Intercambiar el elemento hallado con a_i.
- Incrementar i en una unidad y repetir el proceso hasta que no queden elementos por ordenar.

Ejemplo de ordenamiento por selección



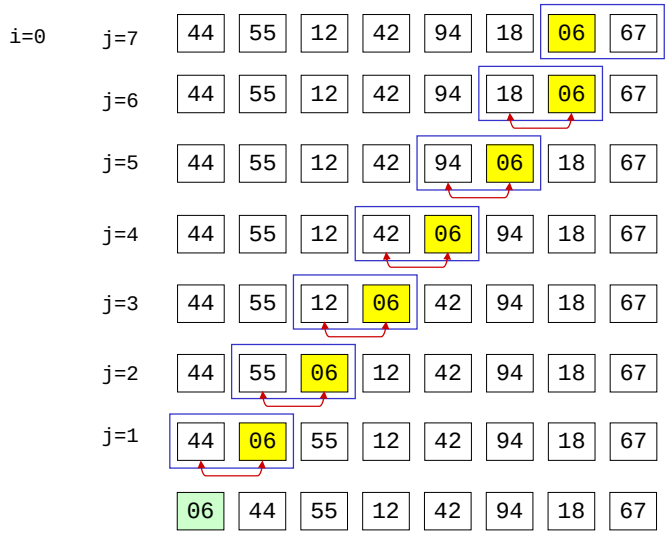
Código para ordenamiento por selección

```
static void selsort(Elem[] array) {
  for (int i=0; i<array.length-1; i++) {
    int lowindex = i;
    for (int j=array.length-1; j>i; j--)
      if (array[j].key() < array[lowindex].key())
              lowindex = j;
    DSutil.swap(array, i, lowindex);
  }
}
```

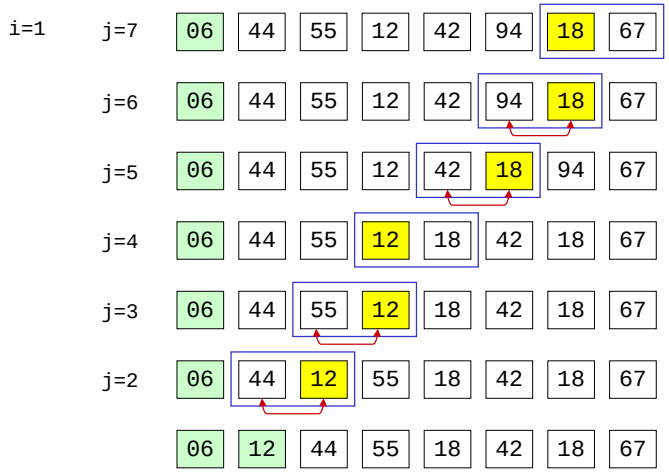

Ordenamiento burbuja

- Recorrer el arreglo secuencialmente varias veces desde el final al comienzo.
- En cada iteración, comparar cada elemento del arreglo con su antecesor, e intercambiarlos si no están en el orden correcto.
- En la iteración *i*-esima (si se parte de i=1), se puede aseverar que el elemento a_{i-1} estará en su posición correcta.
- En la iteración *n-1* todos los elementos estarán ordenados.

Ejemplo de ordenamiento por el método de la burbuja



Ejemplo de ordenamiento por el método de la burbuja (cont.)



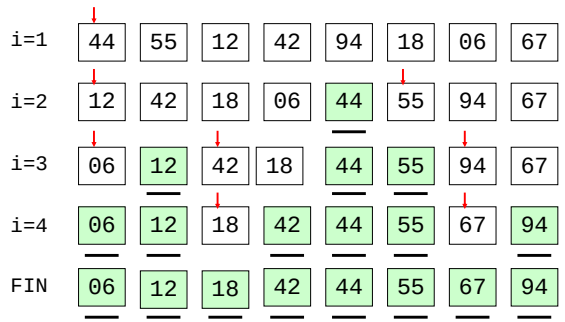
Código para ordenamiento burbuja

```
static void bubsort(Elem[] array) {
  for (int i=0; i<array.length-1; i++)
    for (int j=array.length-1; j>i; j--)
      if (array[j].key() < array[j-1].key())
        DSutil.swap(array, j, j-1);
```


Ordenamiento Quicksort

- Se elige un elemento *x* del arreglo, al que se denomina **pivote**.
- Se reacomodan los registros del arreglo, de modo que los elementos menores que *x* se ubiquen antes que *x*, y los mayores se ubiquen después de él (partición del arreglo).
- Al final de cada partición, se puede asegurar que x está en su posición final *j* dentro del arreglo.
- Repetir el proceso con los arreglos a[1..j-1] y a[j+1..n]

Ejemplo de ordenamiento con Quicksort



↓ = pivote escogido

= elemento en posición correcta

Implementación de QuickSort

```
static void qsort(Elem[] array, int l, int r) {
  int k = partition(array, l, r);
  if (l<k-1) qsort(array, l, k-1);  // Ordena partición izq
  if (k+1<r) qsort(array, k+1, r);  // Ordena partición der
}

static int partition(Elem[] array, int l, int r) {
  int pivot=array[(l+r)/2].key();
  do {
    while ((array[l].key()<pivot) && (l<r)) l++;
    while ((array[r].key()>pivot) && (l<r)) r--;
    if (l<r) DSutil.swap(array, l, r);
  } while( l<r );
return l;
```

Algoritmos de búsqueda

- Dado un conjunto de registros que cuenta con una clave que los diferencia, un algoritmo de búsqueda es un algoritmo que acepta un argumento *A* y trata de hallar un registro que tenga clave *A*.
- La búsqueda puede tener éxito o ser infructuosa.

Búsqueda lineal o secuencial

- Consiste en examinar un registro tras otro hasta hallar uno cuya clave coincida con la clave buscada.
- Si la tabla tiene *n* elementos, el tiempo necesario para hallar un elemento es proporcional al número probable de registros a examinar, que es *n/2*.
- Es un método sencillo, y es razonable usarlo cuando *n* no es muy grande.

Búsqueda secuencial en un arreglo

```
static int sequential(int k, int[] array) {
  // Retorna posición de elemento en arreglo(-1=no se halló)
  int i=0;
  int n=array.length-1;
  while (i<=n) {
    if (k == array[i])
       return i; // Lo halló
    i++;
  }
  return -1; // Valor no está en el arreglo
}
```


Búsqueda en arreglos ordenados

- Supongamos que tenemos un arreglo de registros ordenados por clave.
- **Búsqueda binaria**: comparar la clave buscada con la clave del registro en el medio del arreglo. Si hay éxito, la búsqueda ha terminado. Si no, buscar de la misma manera en la mitad superior o inferior.

Búsqueda binaria en un arreglo

```
static int binary(int K, int[] array) {
  // Retorna posición de elemento en arreglo(-1=no se halló)
  int l=0;
  int r=array.length-1;
  while (l<=r) {          // Para cuando l y r se encuentran
    int i = (l+r)/2;      // Busca en medio de subarreglo
    if (k < array[i])
        r = i-1;            // En mitad izquierda
    if (k == array[i])
        return i;           // Lo halló
    if (k > array[i])
        l = i+1;            // En mitad derecha
  }
  return -1; // Valor no está en el arreglo
}
```


Comentarios sobre búsqueda binaria

- El tiempo necesario para hallar un elemento es proporcional al número probable de registros a examinar, que es $\log_2(n)$ (donde n es el número total de registros).
- En general, es más eficiente que la búsqueda secuencial, en especial cuando n es grande.
- Una situación en que no es conveniente, es en el caso de listas enlazadas.